Viterbi Compiler

User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

UG-VITERBI-12.0

Feedback Subscribe

© 2012 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ISO
9001:2008
Registered

Contents

Chapter 1. About This Compiler

Features . . . 1–1
Release Information . . . 1–2
Device Family Support . . . 1–3
Performance and Resource Utilization . . . 1–4
  Hybrid Architecture . . . 1–4
  Parallel Architecture . . . 1–6
Installation and Licensing . . . 1–7
  OpenCore Plus Evaluation . . . 1–8
  OpenCore Plus Time-Out Behavior . . . 1–8

Chapter 2. Getting Started

Design Flow . . . 2–1
DSP Builder Flow . . . 2–1
MegaWizard Plug-In Manager Flow . . . 2–2
  Parameterize the MegaCore Function . . . 2–3
  Set Up Simulation . . . 2–7
  Generate the MegaCore Function . . . 2–8
Simulate the Design . . . 2–12
Compile the Design . . . 2–12
Program a Device . . . 2–12

Chapter 3. Functional Description

Soft Symbol Inputs . . . 3–1
Encoding Scheme . . . 3–2
State Metrics . . . 3–2
Puncturing Scheme . . . 3–2
Trellis Coded Modulation . . . 3–3
Trellis Termination . . . 3–8
Trellis Initiation . . . 3–8
The Avalon Streaming Interface . . . 3–9
Parameters . . . 3–9
  Architecture Tab . . . 3–10
    BER Estimator . . . 3–10
    Node Synchronization . . . 3–11
  Code Sets Tab . . . 3–12
  Parameters Tab . . . 3–13
    Throughput Calculator . . . 3–13
    Latency Calculator . . . 3–14
  Test Data Tab . . . 3–14
Signals . . . 3–14
  Timing Diagrams . . . 3–18
MegaCore Verification . . . 3–19

Additional Information

Revision History . . . Info–1
How to Contact Altera . . . Info–1
Typographic Conventions . . . Info–2

1. About This Compiler

This document describes the Altera® Viterbi Compiler. The Altera Viterbi Compiler comprises high-performance, soft-decision Viterbi MegaCore functions that implement a wide range of standard Viterbi decoders.

Viterbi decoding (also known as maximum likelihood decoding or forward dynamic programming) is the most common way of decoding convolutional codes by using an asymptotically optimum decoding technique. In its basic form, Viterbi decoding is an efficient, recursive algorithm that performs an optimal exhaustive search.

A convolutional encoder and Viterbi decoder can be used together to provide error correction over a noisy channel, e.g., a communications channel. A convolutional encoder adds redundancy (i.e., extra bits) to a data stream before transmission.

The rate and the generating polynomials describe the convolutional code, hence they describe the convolutional encoder. The rate is the number of transmitted bits per input bit, e.g., a rate 1/2 encodes 1 bit and produces 2 bits for transmission. Similarly, a rate 2/3 encodes 2 bits and produces 3 bits for transmission. A code can be punctured to increase its rate, by deleting some of the encoded bits according to a deterministic pattern.

The generating polynomials denote the convolutional encoder state bits, which are mathematically combined to produce an encoded bit. There is one generating polynomial per encoded bit. The length in bits of the generating polynomial is called the constraint length; systems with higher constraint lengths are generally more robust. However, the complexity of the Viterbi decoder increases exponentially with the constraint length, so it is unusual to find constraint lengths greater than nine.

A noisy transmission channel causes bit errors at the receiver. The Viterbi algorithm finds the most likely sequence of bits that is closest to the actual received sequence. The Viterbi decoder uses the redundancy, which the convolutional encoder imparted, to decode the bit stream and remove the errors.

The receiver can deliver either hard or soft symbols to the Viterbi decoder. A hard symbol is equivalent to a binary ±1. A soft symbol is multi-leveled to represent the confidence in the bit being positive or negative. For instance, if the channel is non-fading and Gaussian, the output of a matched filter quantized to a given number of bits is a suitable soft input. Punctured symbols are indicated with the `eras_sym` input. The Viterbi algorithm has better performance with soft input symbols.

The Viterbi decoder works on blocks of data, or continuous streams. It takes in N symbols at a time for processing, where N is the number of encoded symbols. The traceback length is the number of trellis states processed before the decoder makes a decision on a bit.

Features

The Viterbi Compiler provides two high-performance, area-optimized, soft-decision Viterbi decoder MegaCore functions—the hybrid and parallel architecture. For both MegaCore functions, you can specify the BER estimator, node synchronization, and multiple code sets (including the variable constraint lengths).

The Viterbi Compiler supports the following features:

- High-speed parallel architecture with:
  - Performance of over 250 megabits per second (Mbps)
  - Fully parallel operation
  - Optimized block decoding and continuous decoding
- Low to medium-speed, hybrid architecture
  - Configurable number of add compare and select (ACS) units
  - Memory-based architecture
  - Wide range of performance; wide range of logic area
- Fully parameterized Viterbi decoder, including:
  - Number of coded bits
  - Constraint length
  - Number of soft bits
  - Traceback length
  - Polynomial for each coded bit
- Avalon® Streaming (Avalon-ST) interfaces
- Variable constraint length
- Trellis coded modulation (TCM) option
- Easy-to-use IP Toolbench interface
- DSP Builder ready
- VHDL testbenches to verify the decoder
- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators
- Flexible licensing—use only the features you require
- Support for OpenCore Plus evaluation

Release Information

Table 1–1 provides information about this release of the Viterbi Compiler.

Table 1–1. Viterbi Compiler Release Information (Part 1 of 2)

Item	Description
Version	12.1
Release Date	November 2012
Ordering Code	IP-VITERBI/HS (parallel architecture) IP-VITERBI/SS (hybrid architecture)

Table 1–1. Viterbi Compiler Release Information (Part 2 of 2)

Item	Description
Product IDs	0037 (parallel architecture) 0038 (hybrid architecture)
Vendor ID	6AF7

For more information about this release, refer to the *MegaCore IP Library Release Notes and Errata*.

Altera verifies that the current version of the Quartus® II software compiles the previous version of each MegaCore® function. The *MegaCore IP Library Release Notes and Errata* report any exceptions to this verification. Altera does not verify compilation with MegaCore function versions older than one release."

Device Family Support

Table 1–2 defines the device support levels for Altera IP cores.

Table 1–2. Altera IP Core Device Support Levels

FPGA Device Families	HardCopy Device Families
Preliminary support—The IP core is verified with preliminary timing models for this device family. The IPcore meets all functional requirements, but might still be undergoing timing analysis for the device family. It can be used in production designs with caution.	**HardCopy Companion**—The IP core is verified with preliminary timing models for the HardCopy companion device. The IP core meets all functional requirements, but might still be undergoing timing analysis for the HardCopy device family. It can be used in production designs with caution.
Final support—The IP core is verified with final timing models for this device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.	**HardCopy Compilation**—The IP core is verified with final timing models for the HardCopy device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.

Table 1–3 shows the level of support offered by the Viterbi MegaCore functions to each of the Altera device families.

Table 1–3. Device Family Support (Part 1 of 2)

Device Family	Support
Arria® GX	Final
Arria II GX	Final
Arria II GZ	Final
Arria V GZ	Preliminary
Cyclone®	Final
Cyclone II	Final
Cyclone III	Final
Cyclone III LS	Final
Cyclone IV GX	Final
HardCopy® II	HardCopy Compilation

Table 1–3. Device Family Support (Part 2 of 2)

Device Family	Support
HardCopy III	HardCopy Compilation
HardCopy IV E	HardCopy Compilation
HardCopy IV GX	HardCopy Compilation
Stratix®	Final
Stratix II	Final
Stratix II GX	Final
Stratix III	Final
Stratix IV GT	Final
Stratix IV GX/E	Final
Stratix V	Preliminary
Stratix GX	Final
Other device families	No support

Performance and Resource Utilization

This section shows typical expected performance for different architectures and constraint length, `L`, combinations, and ACS units, `A`, using the Quartus II software, for the following devices:

- Cyclone III (EP3C10F256C6)
- Stratix III (EP3SE50F780C2)
- Stratix IV (EP4SGX70DF29C2X)

Performance largely depends on constraint length, `L`.

Hybrid Architecture

Table 1–4 through Table 1–6 show the performance for the hybrid architecture using the BER option and the following parameters:

`v` = 6 × `L`
`softbits` = 3
`N` = 2

where:

v is the traceback length
L is the constraint length
N is the number of coded bits
A is the number of ACS units

Table 1–4. Performance & Area Utilization for Hybrid Architectures—Cyclone III Devices

Parameters		Combinational LUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
A	L					
1	5	605	387	5	193	19
1	7	825	502	6	197	6
2	7	977	619	6	191	12
4	7	1,259	833	6	185	19
1	9	1,577	922	12	188	1
2	9	1,730	1,047	12	185	3
4	9	2,044	1,277	12	178	6
8	9	2,653	1,723	14	174	11
16	9	3,807	2,585	18	166	17

Table 1–5. Performance & Area Utilization for Hybrid Architectures—Stratix III Devices

Parameters		Combinational ALUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
A	L					
1	5	542	387	5	327	33
1	7	730	502	6	330	10
2	7	889	620	6	341	21
4	7	1,127	833	6	323	32
1	9	1,419	922	12	312	2
2	9	1,582	1,047	12	303	5
4	9	1,896	1,277	12	318	10
8	9	2,466	1,723	14	298	19
16	9	3,487	2,587	18	297	30

Table 1–6. Performance & Area Utilization for Hybrid Architectures—Stratix IV Devices (Part 1 of 2)

Parameters		Combinational ALUTs	Logic Registers	Memory		fMAX (MHz)	Throughput (Mbps)
A	L			ALUTs	M9K		
1	5	548	407	4	4	331	33
1	7	736	526	6	5	328	10
2	7	894	643	6	5	337	21
4	7	1,134	857	6	5	319	32
1	9	1,459	978	24	10	312	2

Table 1–6. Performance & Area Utilization for Hybrid Architectures—Stratix IV Devices (Part 2 of 2)

Parameters		Combinational ALUTs	Logic Registers	Memory		fMAX (MHz)	Throughput (Mbps)
A	L			ALUTs	M9K		
2	9	1,622	1,103	24	10	307	5
4	9	1,936	1,333	24	10	310	10
8	9	2,509	1,780	24	12	285	18
16	9	3,526	2,643	24	16	293	29

Parallel Architecture

Table 1–7 through Table 1–9 show the performance for the parallel architecture with no BER option and the following parameters:

$v = 6 \times L$
$N = 2$

where:

v is the traceback length
L is the constraint length
N is the number of coded bits

Table 1–7. Performance & Area Utilization for Parallel Architecture—Cyclone III Devices

Parameters				Combinational LUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
softbits	L	Optimization	Best State Finder					
7	3	Block	Off	2,218	847	5	184	184
7	2	Continuous	Off	2,048	814	5	181	181
3	3	None	Off	725	436	5	211	211
5	3	None	Off	1,117	574	5	198	198
7	1	None	Off	2,218	964	7	198	198
7	3	None	Off	2,624	1,108	7	187	187
7	4	None	Off	2,825	1,180	7	181	181
3	3	None	On	755	464	5	205	205
5	3	None	On	1,275	720	5	200	200
7	3	None	On	3,307	1,732	7	188	188

Table 1–8. Performance & Area Utilization for Parallel Architecture—Stratix III Devices (Part 1 of 2)

Parameters				Combinational ALUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
softbits	L	Optimization	Best State Finder					
7	3	Block	Off	2,059	848	5	281	281
7	2	Continuous	Off	2,015	816	9	279	279

Table 1–8. Performance & Area Utilization for Parallel Architecture—Stratix III Devices (Part 2 of 2)

Parameters				Combinational ALUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
softbits	L	Optimization	Best State Finder					
3	3	None	Off	548	437	5	327	327
5	3	None	Off	918	574	5	307	307
7	1	None	Off	2,013	970	7	292	292
7	3	None	Off	2,401	1,109	7	285	285
7	4	None	Off	2,596	1,180	7	285	285
3	3	None	On	565	464	5	326	326
5	3	None	On	1,092	723	5	308	308
7	3	None	On	3,082	1,732	7	277	277

Table 1–9. Performance & Area Utilization for Parallel Architecture—Stratix IV Devices

Parameters				Combinational ALUTs	Logic Registers	Memory		fMAX (MHz)	Throughput (Mbps)
softbits	L	Optimization	Best State Finder			ALUTs	M9K		
7	3	Block	Off	2,058	847	--	5	285	285
7	2	Continuous	Off	2,015	815	--	9	292	292
3	3	None	Off	606	523	40	2	345	345
5	3	None	Off	942	608	16	4	316	316
7	1	None	Off	2,044	1,012	24	6	292	292
7	3	None	Off	2,436	1,153	24	6	289	289
7	4	None	Off	754	545	6	5	311	311
3	3	None	On	624	551	40	2	341	341
5	3	None	On	1,115	756	16	4	314	314
7	3	None	On	3,117	1,777	24	6	288	288

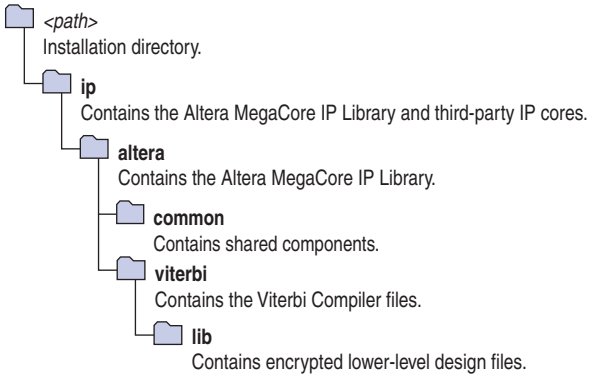
Installation and Licensing

The Viterbi Compiler is part of the MegaCore® IP Library, which is distributed with the Quartus® II software and can be downloaded from the Altera® website, www.altera.com.

For system requirements and installation instructions, refer to the *Altera Software Installation and Licensing* manual.

Figure 1–1 shows the directory structure after you install the Viterbi Compiler, where <*path*> is the installation directory for the Quartus II software. The default installation directory on Windows is **c:\altera\<***version***>**; or on Linux is **/opt/altera<***version***>**.

Figure 1–1. Directory Structure



OpenCore Plus Evaluation

With Altera's free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPP^{SM} megafunction) within your system.
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily.
- Generate time-limited device programming files for designs that include megafunctions.
- Program a device and verify your design in hardware.

You only need to purchase a license for the Viterbi Compiler when you are completely satisfied with its functionality and performance, and want to take your design to production. After you purchase a license, you can request a license file from the Altera website at **www.altera.com/licensing** and install it on your computer. When you request a license file, Altera emails you a **license.dat** file. If you do not have Internet access, contact your local Altera representative.

For more information about OpenCore Plus hardware evaluation, refer to *AN320: OpenCore Plus Evaluation of Megafunctions*.

OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation supports the following operation modes:

- *Untethered*—the design runs for a limited time.
- *Tethered*—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely.

All megafunctions in a device time-out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior might be masked by the time-out behavior of the other megafunctions.

The untethered time-out for the Viterbi Compiler is one hour; the tethered time-out value is indefinite.

Your design stops working after the hardware evaluation time expires and the `decbit` signal remains low.

Design Flow

The Viterbi Compiler supports the following design flows:

- **DSP Builder**: Use this flow if you want to create a DSP Builder model that includes a Viterbi Compiler variation.
- **MegaWizard™ Plug-In Manager**: Use this flow if you would like to create a Viterbi Compiler variation that you can instantiate manually in your design.

This chapter describes how you can use a Viterbi Compiler variation in either of these flows. The parameterization provides the same options in each flow and is described in "Parameterize the MegaCore Function" on page 2–3.

After parameterizing and simulating a design in either of these flows, you can compile the completed design in the Quartus II software.

DSP Builder Flow

Altera's DSP Builder product shortens digital signal processing (DSP) design cycles by helping you create the hardware representation of a DSP design in an algorithm-friendly development environment.

DSP Builder integrates the algorithm development, simulation, and verification capabilities of The MathWorks MATLAB® and Simulink® system-level design tools with Altera Quartus® II software and third-party synthesis and simulation tools. You can combine existing Simulink blocks with Altera DSP Builder blocks and MegaCore function variation blocks to verify system level specifications and perform simulation.

In DSP Builder, a Simulink symbol for the MegaCore function appears in the MegaCore Functions library of the Altera DSP Builder Blockset in the Simulink library browser.

To use the Viterbi Compiler in the MATLAB/Simulink environment, follow these steps:

1. Create a new Simulink model.
2. Select the `viterbi_<version>` block from the MegaCore Functions library in the Simulink Library Browser, add it to your model, and give the block a unique name.
3. Double-click on the `viterbi_<version>` block in your model to display the parameter editor and parameterize the MegaCore function variation. For an example of setting parameters for the Viterbi Compiler, refer to "Parameterize the MegaCore Function" on page 2–3.
4. Click **Finish** in the parameter editor to complete the parameterization and generate your Viterbi Compiler MegaCore function variation. For information about the generated files, refer to Table 2–1 on page 2–11.
5. Connect your Viterbi Compiler variation to the other blocks in your model.

6. Simulate the MegaCore function variation in your DSP Builder model.

For more information about the DSP Builder flow, refer to the *Using MegaCore Functions* chapter in the *DSP Builder User Guide*.

When you are using the DSP Builder flow, device selection, simulation, Quartus II compilation and device programming are all controlled within the DSP Builder environment.

DSP Builder supports integration with SOPC Builder using Avalon® Memory-Mapped (Avalon-MM) master/slave and Avalon Streaming (Avalon-ST) source/sink interfaces.

For more information about these interface types, refer to the *Avalon Interface Specifications*.

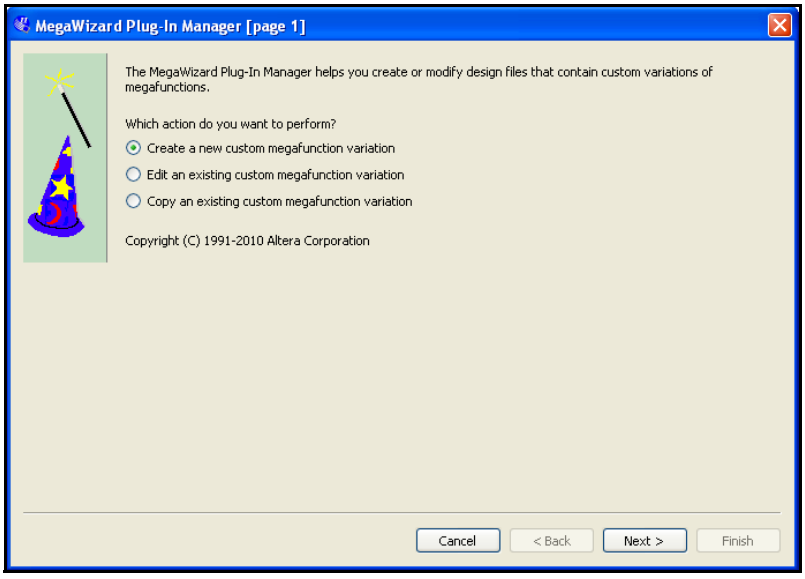
MegaWizard Plug-In Manager Flow

The MegaWizard™ Plug-in Manager flow allows you to customize a Viterbi Compiler MegaCore function, and manually integrate the MegaCore function variation into a Quartus II design.

Follow the steps below to use the MegaWizard Plug-in Manager flow.

1. Create a new project using the **New Project Wizard** available from the File menu in the Quartus II software.
2. Launch **MegaWizard Plug-in Manager** from the Tools menu, and select the option to create a new custom megafunction variation (Figure 2–1).

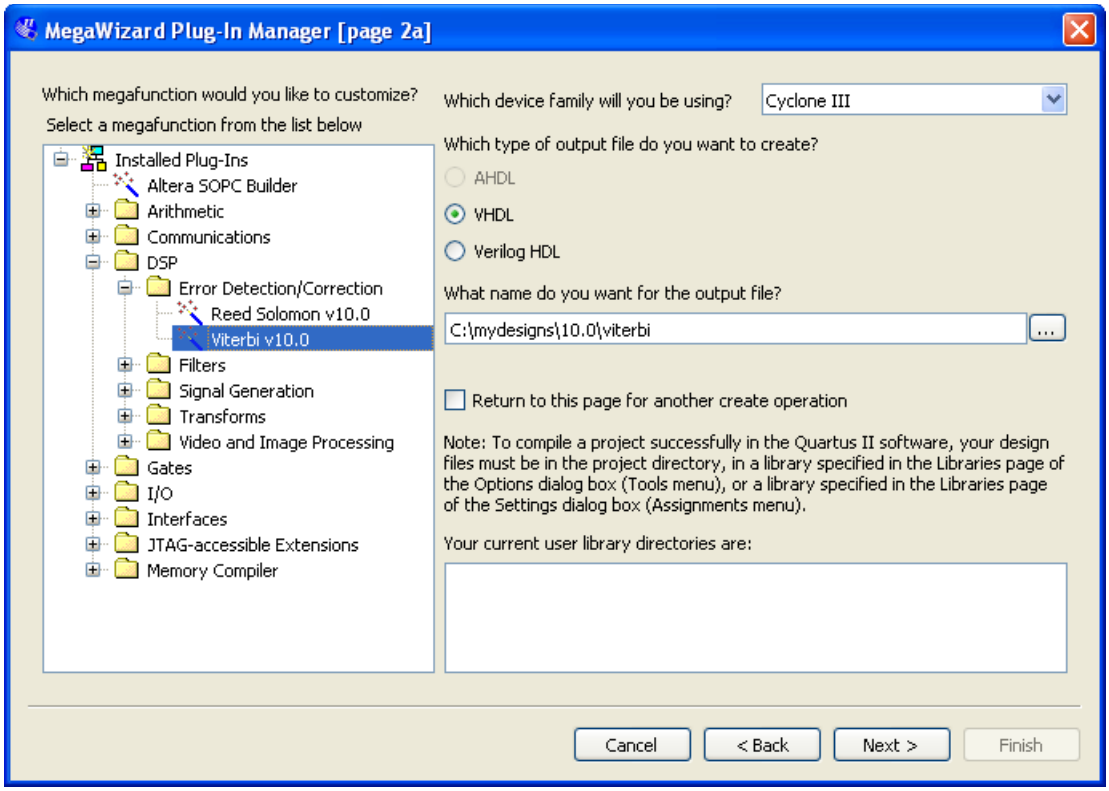
Figure 2–1. MegaWizard Plug-In Manager


MegaWizard Plug-In Manager [page 1]
The MegaWizard Plug-In Manager helps you create or modify design files that contain custom variations of megafunctions.
Which action do you want to perform?
Create a new custom megafunction variation
Edit an existing custom megafunction variation
Copy an existing custom megafunction variation
Copyright (C) 1991-2010 Altera Corporation
Cancel
< Back
Next >
Finish

3. Click **Next** and select **Viterbi** <*version*> from the **DSP >Error Detection/Correction** section in the **Installed Plug-Ins** tab.

4. Verify that the device family is the same as you specified in the **New Project Wizard**.
5. Select the top-level output file type for your design; the wizard supports VHDL and Verilog HDL.
6. The MegaWizard Plug-In Manager shows the project path that you specified in the **New Project Wizard**. Append a variation name for the MegaCore function output files <*project path*>\<*variation name*>. Figure 2–2 shows the wizard after you have made these settings.

Figure 2–2. Selecting the Megafunction


MegaWizard Plug-In Manager [page 2a]
Which megafunction would you like to customize?
Select a megafunction from the list below
Installed Plug-Ins
Altera SOPC Builder
Arithmetic
Communications
DSP
Error Detection/Correction
Reed Solomon v10.0
Viterbi v10.0
Filters
Signal Generation
Transforms
Video and Image Processing
Gates
I/O
Interfaces
JTAG-accessible Extensions
Memory Compiler
Which device family will you be using?
Cyclone III
Which type of output file do you want to create?
AHDL
VHDL
Verilog HDL
What name do you want for the output file?
C:\mydesigns\10.0\viterbi
Return to this page for another create operation
Note: To compile a project successfully in the Quartus II software, your design files must be in the project directory, in a library specified in the Libraries page of the Options dialog box (Tools menu), or a library specified in the Libraries page of the Settings dialog box (Assignments menu).
Your current user library directories are:
Cancel
< Back
Next >
Finish

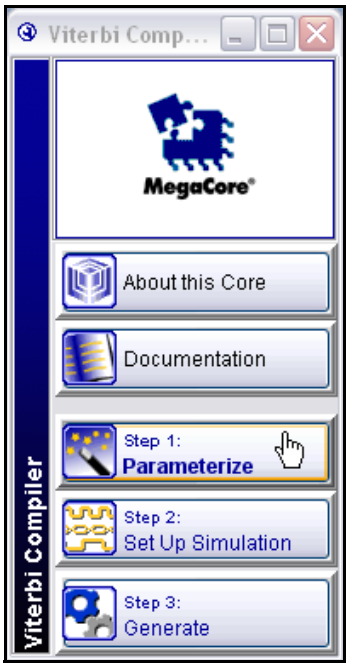
7. Click **Next** to launch IP Toolbench.

Parameterize the MegaCore Function

To parameterize your MegaCore function, follow these steps:

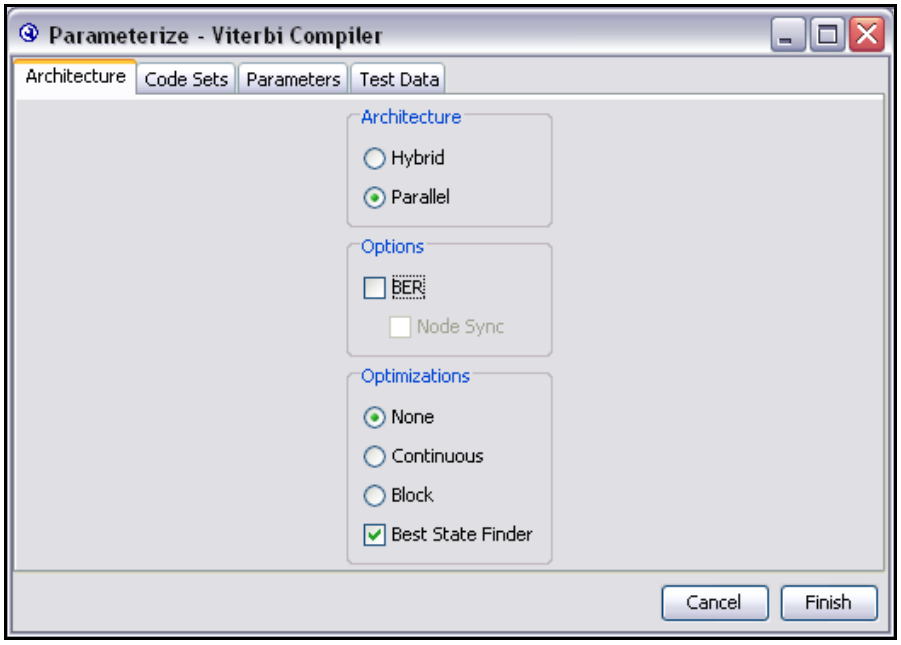
1. Click **Step 1: Parameterize** in IP Toolbench (Figure 2–3).

Figure 2–3. IP Toolbench—Parameterize



2. Select the architecture, either **Hybrid** or **Parallel** (Figure 2–4).

Figure 2–4. Selecting the Architecture

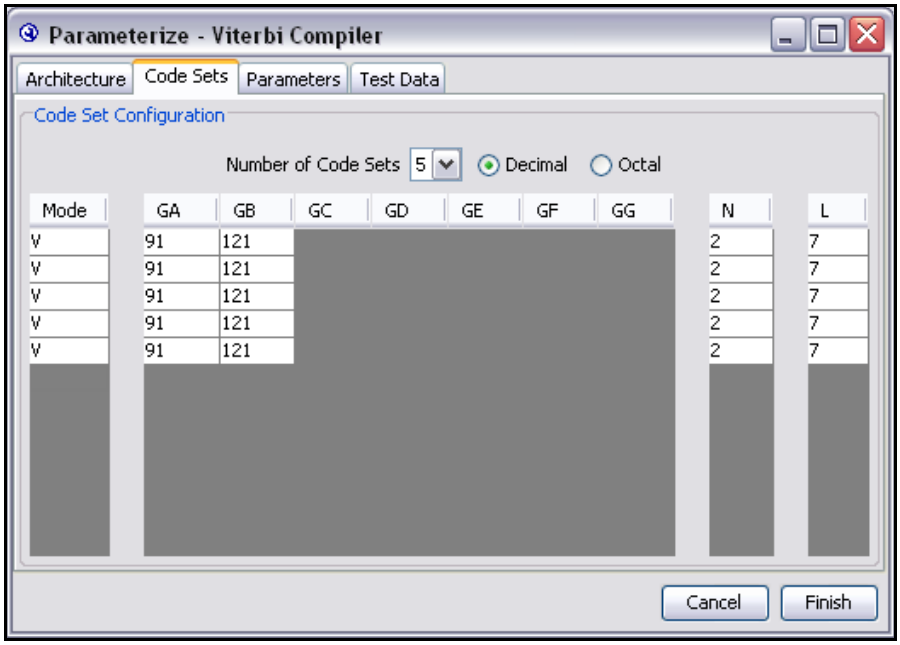


3. Turn on the options that you require. When you turn on **BER**, you can turn on **Node Sync** (Figure 2–4).

For information about BER, refer to "BER Estimator" on page 3–10.

4. For parallel architectures, you can select one of the following optimizations:
   - **None**. The core uses combined continuous and block decoding. With this option only, you can turn on the **Best State Finder** option.
   - **Block**. This option implements a single traceback engine with memory to hold the whole size of the block.
   - **Continuous**. This option implements a fixed traceback length, which reduces the size of the architecture.
5. Click the **Code Sets** tab (Figure 2–5).

Figure 2–5. Code Set Configuration

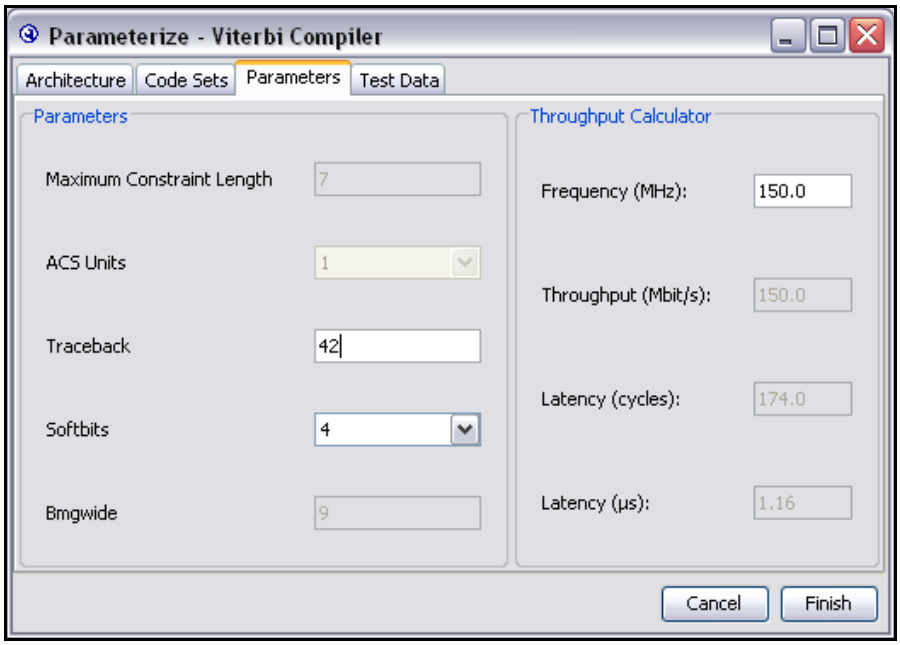


6. Enter the code set information that you require:
   a. Choose the **Number of Code Sets**. Choose a value greater than one, for multiple code sets.
   b. Select **Decimal** or **Octal**.
   c. Choose either Viterbi mode, **V**, or trellis coded modulation (TCM) mode, **T**.
   d. Enter values for the required polynomials **GA**, **GB**, **GC**, **GD**, **GE**, **GF**, and **GG.**
   e. Enter a value for the number of coded bits, **N**.
   f. Enter the constraint length, **L**, for the code set.

For information about multiple code sets, refer to "Code Sets Tab" on page 3–12.

7. Click the **Parameters** tab (Figure 2–6).

Figure 2–6. Choosing the Parameters



8. Choose the parameters that define the specific Viterbi code that you wish to implement:
   a. For the hybrid architecture only, choose the number of **ACS Units**.
   b. Enter the **Traceback** length.
   c. Choose the number of **Softbits**.

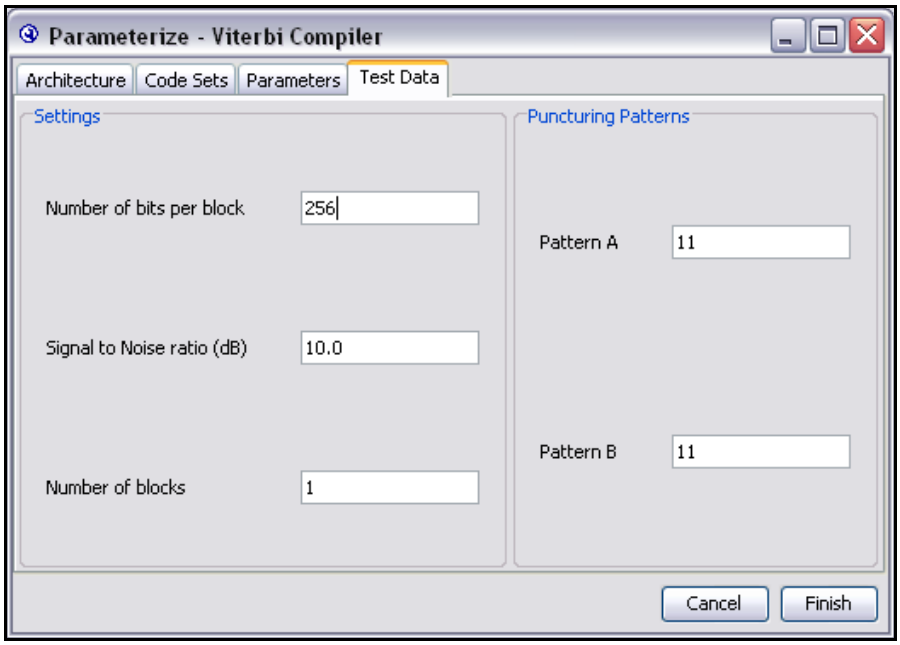
The maximum constraint length **L** that you specified on the **Code Sets** tab is displayed as a read-ony field. The **Bmgwide** field displays the state metric accumulation precision and is calclated from the maximum values of **N**, **L** and **Softbits**.

9. Enter values into the throughput calculator (Figure 2–6). The throughput calculator calculates throughput for specified frequencies.

For information about these parameters, refer to "Parameters Tab" on page 3–13). For the formulae used by the throughput calculator, refer to "Throughput Calculator" on page 3–13. For the formulae used by the latency calculator, refer to "Latency Calculator" on page 3–14.

10. Click the **Test Data** tab (Figure 2–7 on page 2–7).

Figure 2–7. Test Data Settings



11. Enter the test data settings for the testbench:

IP Toolbench generates a VHDL testbench, which can be used in any Altera-supported VHDL simulator. The testbench uses data as specified on this tab.

a. Enter the **Number of bits per block**. The minimum value is equal to the constraint length.

b. Enter the **Signal to Noise ratio** in decbels.

c. Enter the **Number of blocks**.

d. Enter the **Puncturing Patterns**. You can specify de-punctured data for testing.

For information about the test data parameterss, refer to "Test Data Tab" on page 3–14.

12. Click **Finish**.

Set Up Simulation

An IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus II software. The model allows for fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.

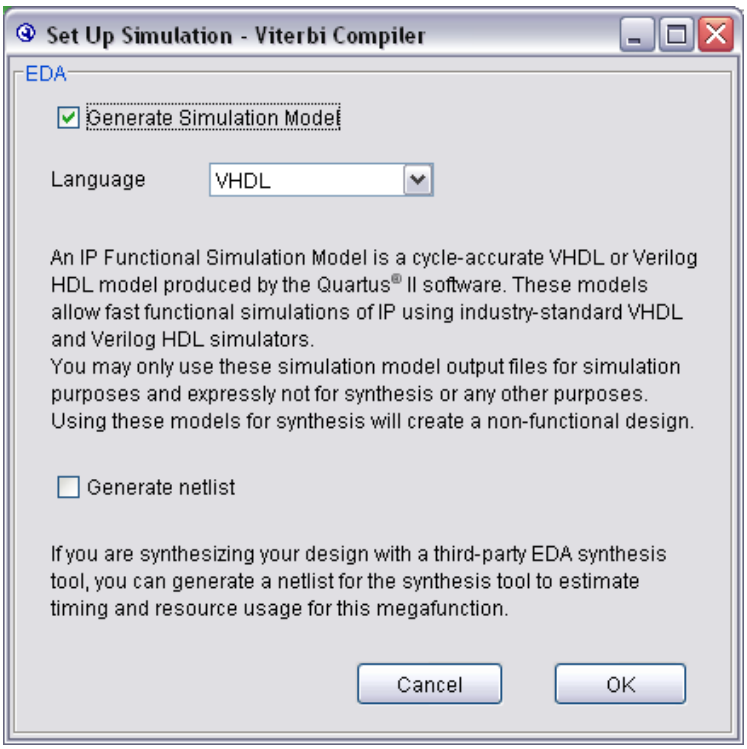
CAUTION

You may only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis will create a nonfunctional design.

To generate an IP functional simulation model (<*variation name*>**.vo** or <*variation name*>**.vho**) for your MegaCore function, follow these steps:

1. Click **Step 2: Set Up Simulation** in IP Toolbench (Figure 2–3 on page 2–4).
2. Turn on **Generate Simulation Model** (Figure 2–8).

Figure 2–8. Generate Simulation Model


Set Up Simulation - Viterbi Compiler
EDA
Generate Simulation Model
Language
VHDL
An IP Functional Simulation Model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus® II software. These models allow fast functional simulations of IP using industry-standard VHDL and Verilog HDL simulators.
You may only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis will create a non-functional design.
Generate netlist
If you are synthesizing your design with a third-party EDA synthesis tool, you can generate a netlist for the synthesis tool to estimate timing and resource usage for this megafunction.
Cancel
OK

3. Choose the required language in the **Language** drop-down box.
4. Some third-party synthesis tools can use a netlist that contains only the structure of the MegaCore function, but not detailed logic, to optimize performance of the design that contains the MegaCore function. If your synthesis tool supports this feature, turn on **Generate netlist**.
5. Click **OK**.

Generate the MegaCore Function

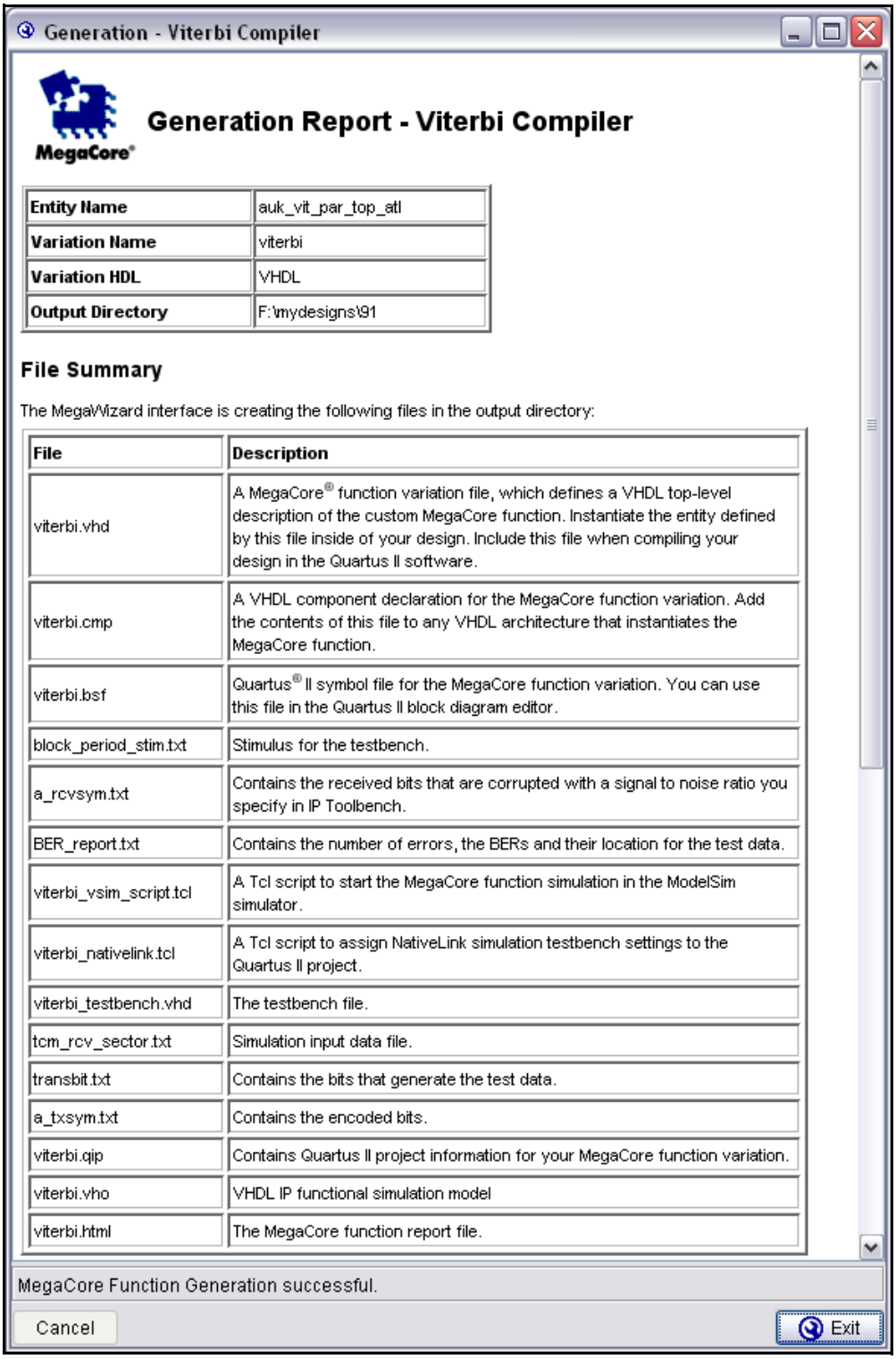
To generate your MegaCore function, follow these steps:

1. Click **Step 3: Generate** in IP Toolbench (Figure 2–3 on page 2–4).

   The generation phase may take several minutes to complete. The generation progress and status is displayed in a report window.

   Figure 2–9 on page 2–10 shows the generation report.

Figure 2–9. Generation Report *(1)*


Generation - Viterbi Compiler
MegaCore
Generation Report - Viterbi Compiler
Entity Name
auk_vit_par_top_atl
Variation Name
viterbi
Variation HDL
VHDL
Output Directory
F:\mydesigns\91
File Summary
The MegaWizard interface is creating the following files in the output directory:
File
Description
viterbi.vhd
A MegaCore® function variation file, which defines a VHDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
viterbi.cmp
A VHDL component declaration for the MegaCore function variation. Add the contents of this file to any VHDL architecture that instantiates the MegaCore function.
viterbi.bsf
Quartus® II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
block_period_stim.txt
Stimulus for the testbench.
a_rcvsym.txt
Contains the received bits that are corrupted with a signal to noise ratio you specify in IP Toolbench.
BER_report.txt
Contains the number of errors, the BERs and their location for the test data.
viterbi_vsim_script.tcl
A Tcl script to start the MegaCore function simulation in the ModelSim simulator.
viterbi_nativelink.tcl
A Tcl script to assign NativeLink simulation testbench settings to the Quartus II project.
viterbi_testbench.vhd
The testbench file.
tcm_rcv_sector.txt
Simulation input data file.
transbit.txt
Contains the bits that generate the test data.
a_txsym.txt
Contains the encoded bits.
viterbi.qip
Contains Quartus II project information for your MegaCore function variation.
viterbi.vho
VHDL IP functional simulation model
viterbi.html
The MegaCore function report file.
MegaCore Function Generation successful.
Cancel
Exit

Note to Figure 2–9:

(1) The **Entity Name** is added automatically. It includes _par for a parallel architecture or _hyb for a hybrid architecture.

Table 2–1 describes the generated files and other files that may be in your project directory. The names and types of files specified in the IP Toolbench report vary based on whether you created your design with VHDL or Verilog HDL.

Table 2–1. Generated Files *(1)*

Filename	Description
<*variation name*>**.bsf**	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<*variation name***>.cmp**	A VHDL component declaration file for the MegaCore function variation. Add the contents of this file to any VHDL architecture that instantiates the MegaCore function.
<*variation name***>.html**	A MegaCore function report file in hypertext markup language format.
<*variation name***>.qip**	A single Quartus II IP file is generated that contains all of the assignments and other information required to process your MegaCore function variation in the Quartus II compiler. You are prompted to add this file to the current Quartus II project when you exit from the MegaWizard.
<*variation name*>**.vho or .vo**	VHDL or Verilog HDL IP functional simulation model.
<*variation name*>**.vhd**, or **.v**	A MegaCore function variation file, which defines a VHDL or Verilog HDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
<*variation name*>**_nativelink.tcl**	Tcl Script that sets up NativeLink in the Quartus II software to natively simulate the design using selected EDA tools.
<*variation name*>**_syn.vhd** or **_syn.v** *(2)*	An optional timing and resource netlist for use in some third-party synthesis tools.
<*variation name*>**_testbench.vhd**	The testbench.
<*variation name*>**_vsim_script.tcl**	Starts the MegaCore function simulation in the ModelSim simulator.
a_rcvsym.txt	Contains the received bits that are corrupted with a signal-to-noise ratio you specify in IP Toolbench.
a_txsym.txt	Contains the encoded bits.
BER_report.txt	Contains the number of errors, the BERs and their location for the test data.
block_period_stim.txt	The testbench stimuli, which change for every block.
tcm_rcv_sector.txt	Contains the sector numbers for TCM codes for decoding the testbench. The file is empty if there are no TCM codes defined.
transbit.txt	Contains the bits that generate the test data.

Notes to Table 2–1:

(1) The <*variation name*> prefix is automatically added by IP Toolbench.

(2) The **_syn.vhd** or **_syn.v** file is only generated when it is enabled in the **Set Up Simulation** page of the IP Toolbench interface.

2. After you review the generation report, click **Exit** to close IP Toolbench. Then click **Yes** on the **Quartus II IP Files** prompt to add the **.qip** file describing your custom MegaCore function to the current Quartus II project.

Refer to the Quartus II Help for more information about the MegaWizard Plug-In Manager.

You can now integrate your custom variation into your system design, simulate, and compile the design.

Simulate the Design

You can perform a simulation in a third-party simulation tool from within the Quartus II software, using NativeLink.

For more information about NativeLink, refer to the *Simulating Altera Designs* chapter in volume 3 of the *Quartus II Handbook*.

You can use the Tcl script file *<variation name>*_**nativelink.tcl** to assign default NativeLink testbench settings to the Quartus II project.

To set up simulation in the Quartus II software using NativeLink, follow these steps:

1. Create a custom variation but ensure you specify your variation name to match the Quartus II project name.
2. Check that the absolute path to your third-party simulator executable is set. On the Tools menu click **Options** and select **EDA Tools Options**.
3. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**.
4. On the Tools menu click **Tcl scripts**. Select the *<variation name>*_**nativelink.tcl** Tcl script and click **Run**. Check for a message confirming that the Tcl script was successfully loaded.
5. On the Assignments menu click **Settings**, expand **EDA Tool Settings** and select **Simulation**. Select a simulator under **Tool Name**.
6. On the Tools menu point to **EDA Simulation Tool** and click **Run EDA RTL Simulation**.

Compile the Design

You can use the Quartus II software to compile your design. Refer to Quartus II Help for instructions on compiling your design.

Program a Device

After you have compiled your design, program your targeted Altera device and verify your design in hardware.

With Altera's free OpenCore Plus evaluation feature, you can evaluate the Viterbi Compiler before you purchase a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model, and produce a time-limited programming file.

For more information about IP functional simulation models, refer to the *Simulating Altera Designs* chapter in volume 3 of the *Quartus II Handbook.*

You can simulate the Viterbi Compiler in your design, and perform a time-limited evaluation of your design in hardware.

For more information about OpenCore Plus hardware evaluation using the Viterbi Compiler, refer to "OpenCore Plus Time-Out Behavior" on page 1–8, and *AN320: OpenCore Plus Evaluation of Megafunctions*.

The Viterbi decoder can decode continuous streams and block streams. It normally operates in continuous mode.

In continuous mode, the decoder waits until it has processed a number of symbols greater than the traceback length. When the decoder has traced back the number of bits indicated in the traceback length, it starts delivering output bits. This behavior is repeated as long as it remains in continuous mode but changes when the end of packet (EOP) signal is aserted. The decoder then switches to block mode, starting traceback from the last symbol or state. The `tr_init_state` signal indicates the end state that starts the traceback operation. For block decoding it is recommended to indicate the end state of the tail bits (usually zero) and set the `tb_type` port to 1.

Soft Symbol Inputs

The number of soft decision bits per symbol, `softbits`, represent $2^{softbits-1}$ soft 0s and $2^{softbits-1}$ soft 1s. The input values represent received signal amplitudes. If the input is in log-likelihood format, a transformation is required and you must use extra softbits to retain signal integrity. Depunctured values are separately marked. The decoder allows a hard-decision input when `softbits` = 1.

Table 3–1 shows an example of the soft symbol input representation, when `softbits` = 3.

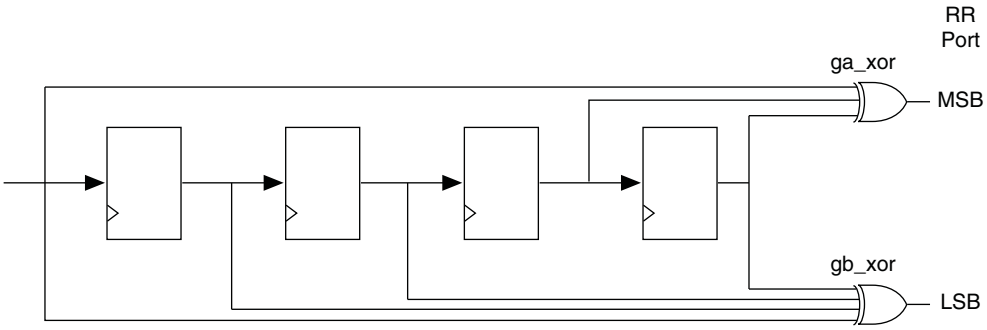
Table 3–1. Soft Symbol Input Representation

Soft Symbol	Meaning
011	Strongest '0'
010	Strong '0'
001	Weak '0'
000	Weakest '0'
111	Weakest '1'
110	Weak '1'
101	Strong '1'
100	Strongest '1'

Encoding Scheme

Figure 3–1 on page 3–2 shows a convolutional encoder with parameters L = 5, N = 2 and polynomials GA = 19 and GB = 29. GA in decimal is 19, which is equal to 10011 in binary. The most significant bit of the binary representation is the connection at the input data bit; the least significant bit represents the connection at the end of the shift register chain. The XOR function implements the modulo-2 adding operation.

Figure 3–1. Encoding Scheme



State Metrics

The Viterbi decoder state metrics are accumulative not Euclidean and are based on maximum metrics rather than minimum metrics. As the metrics grow, they must be normalized to avoid overflow. When a normalization occurs the decoder subtracts $2^{(bmgwide-1)}$ from all metrics and increases the normalization register by +1.

The total metric value for the best path = (number of normalizations) × ($2^{(bmgwide-1)}$) + bestmet.

The total metric value for the best path, the number of symbols processed, and the number of errors in the bit error rate (BER) block indicate the quality of the channel and whether you have a suitable value for softbits. The state that has that best metric is given in the output bestadd.

Puncturing Scheme

Both parallel and hybrid architectures support external puncturing. All punctured codes shown are based on a mother code of rate 1/2. For external depuncturing you must depuncture the received data stream external to the decoder, and input the data into the decoder n symbols at a time.

Table 3–2 shows some possible puncturing schemes, which can be defined, and their rate.

Table 3–2. Some Puncturing Schemes (Part 1 of 2)

Punctured Rate	Puncturing Scheme		
	Bit (1)	Multiplier	
2/3	CA	1	0
	CB	1	1

Table 3–2. Some Puncturing Schemes (Part 2 of 2)

Punctured Rate	Puncturing Scheme							
	Bit (1)	Multiplier						
3/4	CA	1	0	1				
	CB	1	1	0				
4/5	CA	1	0	0	0			
	CB	1	1	1	1			
5/6	CA	1	0	1	0	1		
	CB	1	1	0	1	0		
6/7	CA	1	0	0	1	0	1	
	CB	1	1	1	0	1	0	
7/8	CA	1	1	1	1	0	1	0
	CB	1	0	0	0	1	0	1

Note to Table 3–2:

(1) CA refers to the most significant (first transmitted bit, first received symbol); CB refers to the least significant (last transmitted bit, last received symbol).

Trellis Coded Modulation

Trellis coded modulation (TCM) combines modulation and encoding processes to achieve better efficiency without increasing the bandwidth.

Bandwidth-constrained channels operate in the region where $R/W > 1$, where R = data rate and W = bandwidth available. For such channels digital communication systems use bandwidth efficient multi-level phase modulation. For example, phase shift keying (PSK), phase amplitude modulation (PAM), or quadrature amplitude modulation (QAM).

When TCM is applied to a bandwidth-constrained channel, a performance gain results without expanding the signal bandwidth. An increase in the number of signal phases from four to eight requires approximately 4 dB in additional signal power to maintain the same error rate. Hence, if TCM is to provide a benefit, the performance gain of the rate 2/3 code must overcome this 4-dB penalty. If the modulation is an integral part of the encoding process and is designed in conjunction with the code to increase the minimum Euclidian distance between the pairs of coded signals, the loss from the expansion of the signal set is easily overcome and significant coding gain is achieved with relatively simple codes.

Any bandwidth-constrained system benefits from this technique, for example, satellite modem systems.

The Altera Viterbi decoder in TCM mode only supports `N` = 2 (only mother code rates of 1/2).

Consider the use of the 1/2 rate convolutional code (Figure 3–2 on page 3–4) to encode one information bit while the second information bit is left uncoded. When used in conjunction with an eight-point signal constellation, for example, eight-PSK, the two bits select one of the four subsets in the signal constellation, while the remaining information bit selects one of the two points within each subset.

Figure 3–2. Half-Rate Convolutional Code

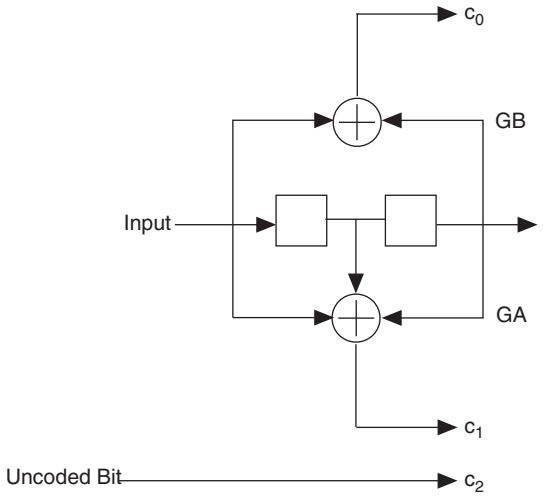
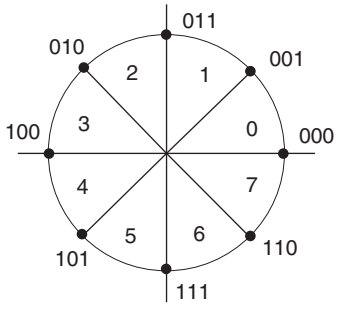


Figure 3–2 shows the mapping of the coded bits and sector numbers. The specific mapping is not important. Other mappings can be devised by permutating subsets in a way that preserves the main property of increased minimum distance among the subsets. IP Toolbench and the testbench create TCM with the mapping shown in Figure 3–2. However, it is possible to create any other mapping, including symbol mappings for 8-PSK, 16-PSK and others.

If you create another mapping, you must correctly connect the branch metrics created outside the MegaCore function to the input ports and correctly configure the polynomials GA and GB for the trellis generation.

Figure 3–3. Mapping of Coded Bits & Sector Numbers



The four-state trellis is the trellis for the 1/2 rate convolution encoder with the addition of parallel paths in each transition to accommodate the uncoded bit c_2. Thus, the decoder uses the coded bits (c_1, c_0) to select one of the four subsets that contain two signal points each, and uses the uncoded bit to select one of the two signal points within each subset.

Figure 3–4. Four-State Trellis

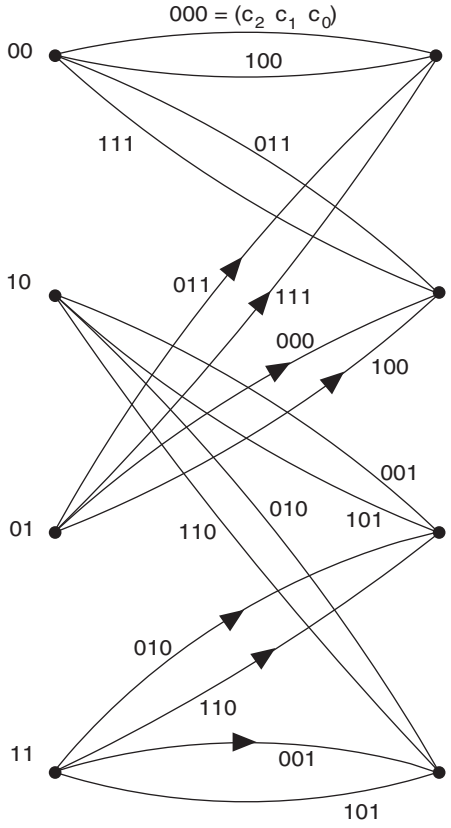


Figure 3–5 shows one implementation of the Viterbi decoder as a Trellis decoder. The decoder processes a symbol upon arrival to obtain four branch metric values and a sector number. The branch metrics enter the Viterbi decoder in trellis mode and the bit that is encoded is obtained. This bit stream is then re-encoded and the output of this encoder is used in conjunction with the sector number information to retrieve the uncoded bit. All the logic is implemented in the provided testbench.

The branch metric values and sector number values are generated by IP Toolbench, so there is no logic to create those values. The testbench reads the sector number when it is needed, hence there is no delay functionality for that, nor is there rotation. The data created by IP Toolbench has no phase error introduced so the phase is aligned. However, in a real system, you must calculate the phase.

For a TCM code the BER block does not produce a meaningful output (`numerr`), because the BER block does not compute errors at the input for TCM codes.

Figure 3–5. Implementation of the Viterbi Decoder as a Trellis Decoder

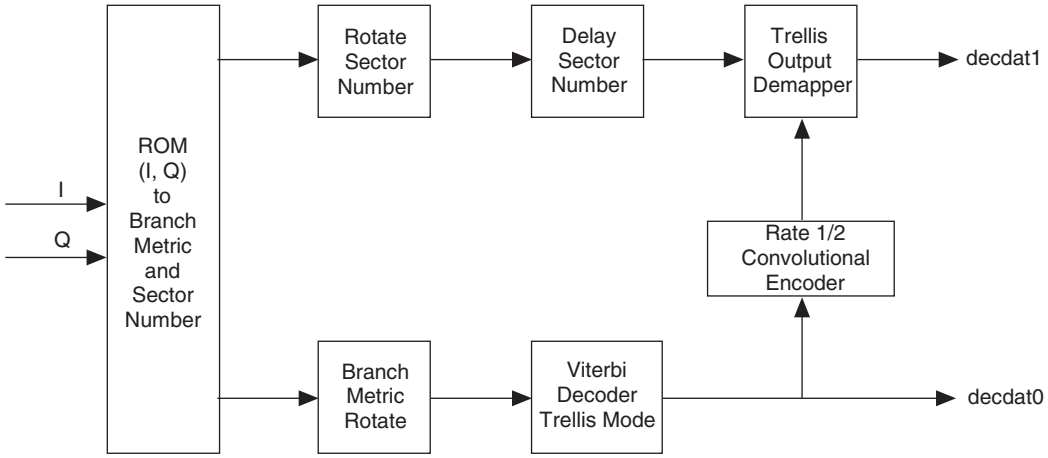
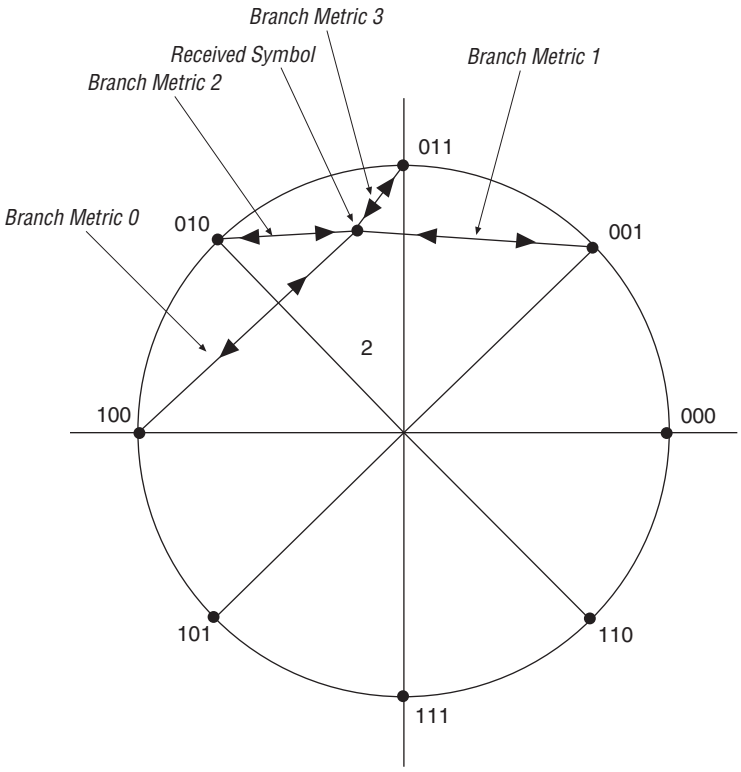


Figure 3–6 shows the conversion of a received symbol into four branch metrics and a sector number. The decoder calculates the distances to the nearest four symbol points as an unsigned number in the range 0...00 to 1...11 (number of softbits). Where the range is equal to the radius of the symbol map. Because the decoder works with accumulative metrics (not Euclidean metrics), the decoder inverts these distances (000 becomes 111; 001 becomes110).

Figure 3–6. Conversion of Received Symbol into Four Branch Metrics



For example, Figure 3–6 shows a received symbol that has landed in sector number 2 with the following distances to the four nearest symbol map points:

- 1111
- 1101
- 1011
- 0001

Where the distance of the radius for 4 softbits is 1111. The distance are inverted to obtain the following branch metrics:

- Branch metric 0 = 0000
- Branch metric 1 = 0010
- Branch metric 2 = 0100
- Branch metric 3 = 1110

The decoder uses the coded bits (c_1, c_0) to select the branch metric number, which is used to decide where to connect the branch metrics to the `rr` input of the Viterbi decoder. Branch metric 3 goes to the most significant bits (MSB) of `rr`; branch metric 0 goes to the least significant bits (LSB) of `rr`.

Trellis Termination

Block decoding requires the implementation of a technique to properly decode the last bits of the block. The technique adapts to whatever is happening in the convolutional encoder. Two techniques are described.

With the first technique, the convolutional encoder is fed with a block and then terminated with (`L` – 1) bits taken from the end of the block. These bits are unknown. The initial state of the convolutional encoder is set with the last (`L` – 1) information bits.

This technique, known as "tail-bitting", is decoded by replicating the block at the decoder or double feeding the block into the decoder. By decoding in the middle point, the trellis is forced into the state that is both the initial state and the end state. From the first decoding block, you can take the last half of the block; from the second decoded block (or second pass through the decoder), you can obtain the first half of the bits of the block.

In tail-bitting technique, the block size must be large enough to train the decoder, otherwise there is BER loss.

With the second technique, the convolutional encoder is initialized to zero. So the initial state of the trellis is known to be zero. The last (`L` – 1) bits to the convolutional encoder are known. They serve the purpose of bringing the convolutional encoder to a known end state. The decoder then uses this information to set the end state of the trellis with `tr_init_state`.

The `tr_init_state` signal is derived from the last (`L` – 1) bits of the block in reverse order.

For example, for a block that ends in:

```
...000101
```

If `L` = 5 and the last (`L` – 1) = 4 bits are known, `tr_init_state` is set as `0101`, which reversed and in binary is `1010`, or 10 in decimal.

IP Toolbench generates `tr_init_state` as if the last (`L` – 1) bits of each block are known.

Trellis Initiation

The parallel decoder always starts its trellis from state zero for a new block. The hybrid however allows you to set the initial state (usually zero) with `bm_init_state`. This signal has range from 0 to $2^{(L-1)-1}$, which are the trellis states.

The `bm_init_value` signal initiates the state metric of the state indicated by `bm_init_state`. All other states are initialized with zero. The appropriate value for this port is approximately $2^{(bmgwide-2)}$ or any value between $2^{(N+softbits)}$ to $2^{(bmgwide-1)}$.

In continuous mode, the state metrics are never reset, which creates a possible difference if the same block of data is sent several times. The first time, the state metrics are set such that the state metric for state 0 is 0, and all others infinity, based on the assumption that the first state is always state 0. For any future blocks, the state metrics contains whatever they had when the previous block ended.

The Avalon Streaming Interface

The Avalon® Streaming (Avalon-ST) interface defines a standard, flexible, and modular protocol for data transfers from a source interface to a sink interface and simplifies the process of controlling the flow of data in a datapath. The Avalon-ST interface signals can describe traditional streaming interfaces supporting a single stream of data without knowledge of channels or packet boundaries. Such interfaces typically contain data, ready, and valid signals. The Avalon-ST interface can also support more complex protocols for burst and packet transfers with packets interleaved across multiple channels. The Avalon-ST interface inherently synchronizes multi-channel designs, which allows you to achieve efficient, time-multiplexed implementations without having to implement complex control logic.

The Avalon-ST interface supports backpressure, which is a flow control mechanism, where a sink can signal to a source to stop sending data. The sink typically uses backpressure to stop the flow of data when its FIFO buffers are full or when there is congestion on its output. When designing a datapath, which includes the Viterbi MegaCore function, you may not need backpressure if you know the downstream components can always receive data. You may achieve a higher clock rate by driving the source ready signal `source_rdy` of the Viterbi high, and not connecting the sink ready signal `sink_rdy`.

For more information about the Avalon-ST interface, refer to the *Avalon Streaming Interface Specifications*.

Parameters

This section contains information about the following parameters and product options, which can be set in IP Toolbench (refer to "Parameterize the MegaCore Function" on page 2–3):

- Architecture Tab
- Parameters Tab
- Code Sets Tab
- Test Data Tab

Architecture Tab

Table 3–3 shows the options that can be set in the **Architecture** tab

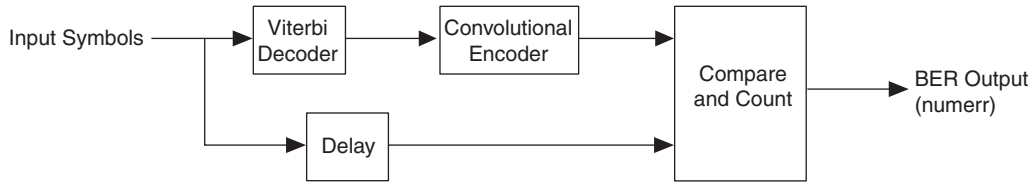
Table 3–3. Architecture Tab Options

Parameter	Value	Description
Hybrid or Parallel	–	Selects the hybrid or parallel architecture.
BER	On or Off	Specifies the BER estimator option, refer to "BER Estimator" on page 3–10.
Node Sync	On or Off	Specifies the node synchronization option (only available when **BER option** is on).
Optimizations	None, Continuous, or Block	Specifies the optimization for the parallel decoder. if you select **None** you can turn on **Best State Finder**. However, to use less logic, turn off **Best State Finder**.

BER Estimator

Figure 3–7 shows a block diagram of the BER estimator.

Figure 3–7. BER Estimator

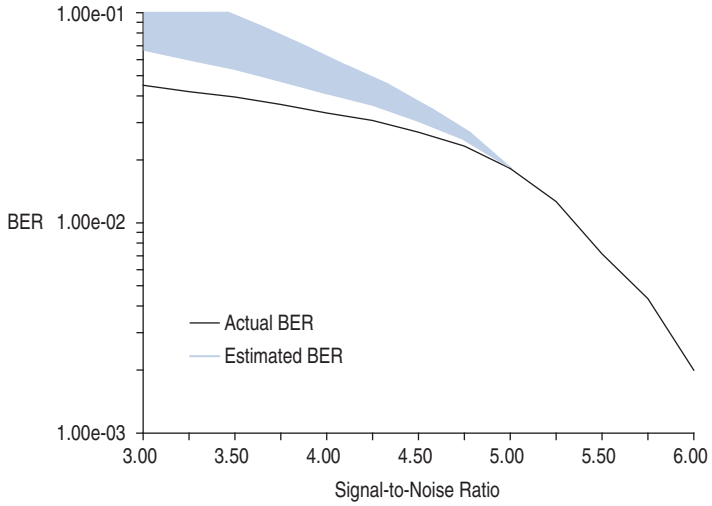


The BER estimator option uses a re-encode and compare approach for estimating the number of errors in the input data. In cases where the signal-to-noise ratio is sufficiently high to allow the decoder to decode an error-free output, the BER estimation is very close to the actual channel BER.

When the decoder is not decoding an error-free output, the estimated BER is higher and more random than the actual channel BER, which introduces a degree of uncertainty directly proportional to the output errors (Figure 3–8 on page 3–11).

For a TCM code, the BER block does not produce a meaningful output (`numerr`) because the BER block does not compute errors at the input for TCM codes.

Figure 3–8. Graph comparing Actual BER with Estimated BER

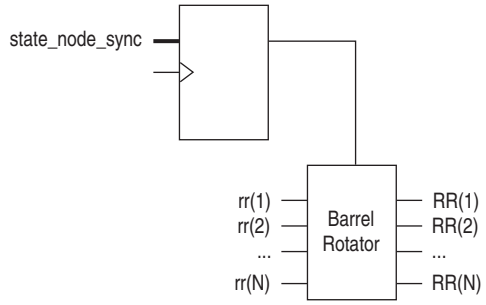


Node Synchronization

If you are not using external synchronization, you may not know the order of your `N` bits. The node synchronization option allows you to rotate the `rr` inputs until the decoder is in synchronization. To use node synchronization, you observe the BER and keep changing `state_node_sync` to rotate the `rr` inputs until you get the correct value for the BER.

Figure 3–9 shows the node synchronization block diagram.

Figure 3–9. Node Synchronization



Note:

(1) The barrel rotator is only implemented if you select the node synchronization option.

The following equation represents node synchronization:

$$RR[i] = rr[((state\_node\_sync + i - 1) \bmod N) + 1]$$

where i is 1 to `N`.

`RR` and `rr` are treated as an array of width `N` of busses `softbits` wide. The range of valid values for `state_node_sync` is 0 to (`N` – 1).

Code Sets Tab

Table 3–4 shows the options that can be set in the **Code Sets** tab.

Table 3–4. Code Sets Tab Options

Parameter	Value	Description
Number of Code Sets	1 to 8	The Viterbi Compiler supports multiple code definitions. The multiple code set option allows up to eight code sets, where a code set comprises a code rate and associated generating polynomials.
Decimal or Octal	–	Decimal or octal base representation for the generator polynomials. The design file representation is decimal, but you have the option of entering in either decimal or octal base.
Mode	V or T	Viterbi (V) or TCM mode (T).
GA, GB, GC, GD, GE, GF, GG (1)	–	The generator polynomials. If the multiple code set option is used, a different set of polynomials is entered in the respective gi group. IP Toolbench provides default values that can be overwritten by any valid polynomial. (The wizard does not check whether the entered values are valid.)
Number of coded bits. (N)	2 to 7 (hybrid) 2 to 4 (parallel) (2)	For every bit to be encoded, N bits are output. With the multiple code set option there are up to 5 different N parameters, which can be in any order.
Constraint length (L)	3 to 9	The constraint length. Defines the number of states in the convolutional encoder, where number of states = $2^{(L-1)}$. You can choose different values of L for each code set.

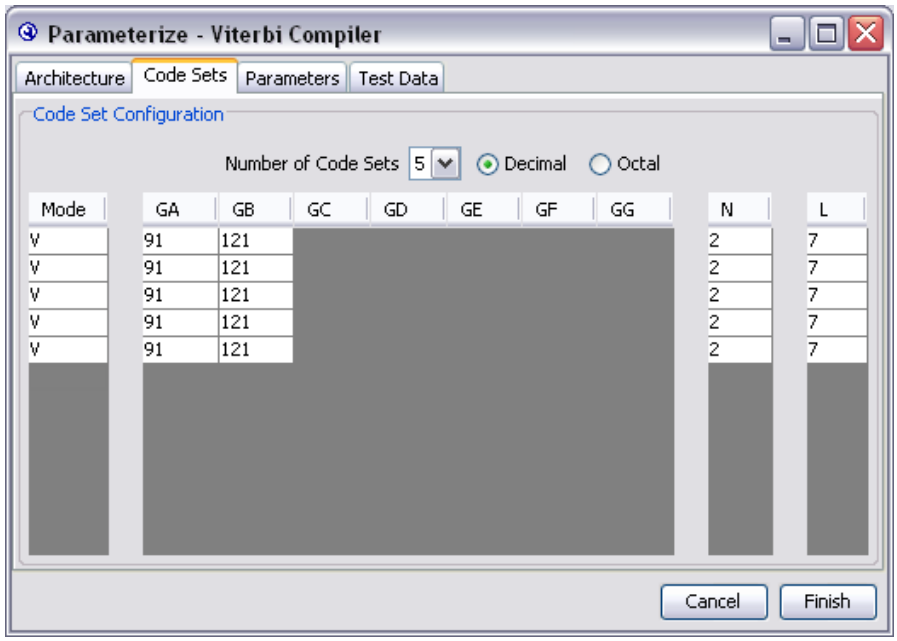
Notes to Table 3–4:

(1) For the parallel architecture, only GA, GB, GC, and GD are used.

(2) Valid only for Viterbi mode. For TCM mode only N = 2 is supported.

Figure 3–10 shows the **Code Sets** tab.

Figure 3–10. Code Sets



For multiple code sets, the first code definition corresponds to the first line and is selected with `sel_code` input = 0; the second line is selected with `sel_code` = 1; the third with `sel_code` = 2 and so on. For each code definition you can select `N`, the polynomials, the constraint length `L`, and the mode (Viterbi or TCM). You can mix different constraint lengths with different TCM and Viterbi modes. The test data, which IP Toolbench creates, tests each of the code definitions. You can see these tests in the simulation with the testbench or if you look at the **block_period_stim.txt** file.

In hybrid mode, for constraint lengths of 3 and 4, the bitwidth of `tr_init_state` is 4, but the MegaCore function ignores the redundant higher bits.

For multiple constraint lengths, some of the last decoded bits may be incorrect, as a result of the Viterbi algorithm. To avoid this effect, give a lower BER, and reduce the probability of being on the wrong trellis path, set **Optimization** to **None** and turn on **Best State Finder**.

Parameters Tab

Table 3–5 shows the options that can be set in the **Parameters** tab.

Table 3–5. Parameters Tab Options

Parameter	Value	Description
Maximum Constraint length (L_{MAX})	5 to 9 (hybrid) 3 to 9 (parallel)	The maximum constraint length L_{MAX}. (Refer also to "Code Sets Tab" on page 3–12.)
ACS Units (`A`)	1, 2, 4, 8, or 16	The number of ACS units, which adds a degree of parallelism (hybrid architecture only). The range of values available depends upon the value of maximum constraint length L_{MAX}.
Traceback (`v`)	8 (minimum)	The traceback length, which is the number of stages in the trellis that are traced back to obtain a decoded bit. It is typically set to 6 × `L` for unpunctured codes, and up to 15 × `L` for highly punctured codes.
Softbits (`softbits`)	1 to 16	The number of soft decision bits per symbol. When softbits is set to 1 bit, the decoder acts as a hard decision decoder, and still allows for erased symbols to be entered using the `eras_sym` input. (Refer to "Soft Symbol Inputs" on page 3–1.)
Bmgwide	–	The precision of the state metric accumulation. (Refer to "State Metrics" on page 3–2) IP Toolbench selects and displays the optimum value, which depends on N_{MAX}, L_{MAX} and, `softbits`.

Throughput Calculator

The throughput calculator uses the following formulae:

$$\text{Hybrid throughput} = f_{MAX}/Z$$

where:

$Z = 10$, if $\log_2 C = 3$
$Z = 2^{\log_2 C}$, if $\log_2 C > 3$
$\log_2 C = L_{MAX} - 2 - \log_2 A$
L_{MAX} is the maximum constraint length
A is ACS units

$$\text{Parallel throughput} = f_{MAX}$$

Latency Calculator

The latency calculator gives you an approximate indication of the latency of your Viterbi decoder. Latency is the number of clock cycles it takes for the data to be processed and output. It is measured from the first symbol to enter the MegaCore function (`sink_sop`) up to the first symbol to leave the MegaCore function (`source_sop`). The latency depends on the parameters.

For the precise latency, perform simulation.

The latency calculator uses the following formula for the hybrid architecture:

Number of clock cycles = $Z \times V$

where:

V is the traceback length value that is in the input `tb_length`
$Z = 10$, if $\log_2 C = 3$
$Z = 2^{\log_2 C}$, if $\log_2 C > 3$
$\log_2 C = l_{max} - 2 - \log_2 A$, where $A$is ACS units

For the parallel architecture the number of clock cycles is approximately $4V$

Test Data Tab

Table 3–6 shows the options that can be set in the **Test Data** tab.

Table 3–6. Test Data

Parameter	Description
Number of bits per block	The number of bits per block. The number of bits per block × the number of blocks must be less than 50,000,000.
Signal to noise ratio (dB)	The signal to noise ratio, which must be between 1 and 100.
Number of blocks	The number of blocks. The number of bits per block × the number of blocks must be less than 50,000,000.
Pattern A	Enter the puncturing pattern A.
Pattern B	Enter the puncturing pattern B.

Signals

The Viterbi decoder uses the Avalon Streaming (Avalon-ST) interface for its data input and output. The input is an Avalon-ST sink and the output is an Avalon-ST source. The Avalon-ST interface `READY_LATENCY` parameter is set to 1.

For more information about the Avalon-ST interface, refer to the *Avalon Interface Specifications*.

Figure 3–11 shows the Viterbi decoder Avalon-ST interfaces.

Figure 3–11. Avalon-ST Interface

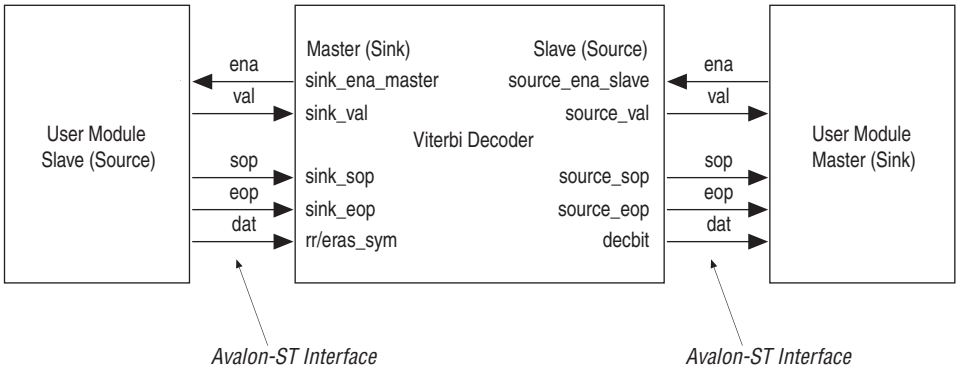


Table 3–7 shows the global signals.

Table 3–7. Global Signals

Signal Name	Description
clk	The main system clock. The whole MegaCore function operates on the rising edge of clk.
reset	Reset. The entire decoder is asynchronously reset when reset is asserted high. The reset signal resets the entire system. The reset signal must be deasserted synchronously with respect to the rising edge of clk.

Table 3–8 shows the Avalon-ST sink signals.

Table 3–8. Avalon-ST Sink Signals (Part 1 of 2)

Signal Name	Avalon-ST Name	Direction	Description
sink_rdy	ready	Output	Data transfer enable signal. sink_rdy is driven by the interface sink and controls the flow of data across the interface. sink_rdy behaves as a read enable from sink to source. When the source observes sink_rdy asserted on the clk rising edge, it can drive the Avalon-ST data interface signals and assert sink_val as early as the next clock cycle, if data is available. In the hybrid architecture, sink_rdy is asserted for one clock cycle at a time. If data is not available at the time, you have to wait for the next sink_rdy pulse. Previously called sink_ena_master.
sink_val	val	Input	Data valid signal. sink_val indicates the validity of the data signals. sink_val is updated on every clock edge where sink_rdy is sampled asserted, and holds its current value along with the dat bus where sink_rdy is sampled deasserted. When sink_val is asserted, the Avalon-ST data interface signals are valid. When sink_val is deasserted, the Avalon-ST data interface signals are invalid and must be disregarded. To determine whether new data has been received, the sink qualifies the sink_val signal with the previous state of the sink_rdy signal.

Table 3–8. Avalon-ST Sink Signals (Part 2 of 2)

Signal Name	Avalon-ST Name	Direction	Description
sink_sop	sop	Input	Start of packet (block) signal. sop delineates the packet boundaries on the rr bus. When sink_sop is high, the start of the packet is present on the rr bus. sink_sop is asserted on the first transfer of every packet This signal applies to block decoding only.
sink_eop	eop	Input	End of packet (block) signal. sink_eop delineates the packet boundaries on the rr bus. When sink_eop is high, the end of the packet is present on the dat bus. sink_eop is asserted on the last transfer of every packet. This signal applies to block decoding only.
rr (1)	dat (2)	Input	Data input, which takes in n symbols, each softbits wide per clock. "Encoding Scheme" on page 3–2 describes the correspondence of the input symbols with the output of a convolutional encoder. For the mappings of the individual soft symbols, refer to Table 3–1.
eras_sym[Nmax:1]	dat (2)	Input	When asserted, eras_sym Indicates an erased symbol.

Notes to Table 3–8:

(1) In TCM mode the rr width is (2^N × softbits:1); in Viterbi mode the rr width is (nmax × softbits:1).

(2) Both rr and eras_sym are seen as Avalon-ST dat inputs.

Table 3–9 shows the Avalon-ST source signals.

Table 3–9. Source Signals

Signal	Avalon-ST Name	Direction	Description
source_rdy	ready	Input	Data transfer enable signal. source_rdy is driven by the sink interface and used to control the flow of data across the interface. ena behaves as a read enable from sink to source. When the source observes source_rdy asserted on the clk rising edge it drives, on the following clk rising edge, the Avalon-ST data interface signals and asserts source_val. The sink captures the data interface signals on the following clk rising edge. If the source is unable to provide new data, it deasserts source_val for one or more clock cycles until it is prepared to drive valid data interface signals. Previously called source_ena_slave.
source_val	val	Output	Data valid signal. source_val is asserted high for one clock cycle, whenever there is a valid output on the decbit signal.
source_sop	sop	Output	Start of packet (block) signal. if you select continuous optimization, this signal is left open and you must remove it from the testbench.
source_eop	eop	Output	End of packet (block) signal. if you select continuous optimization, this signal is left open and you must remove it from the testbench.
decbit	dat	Output	The decbit signal contains output bits when source_val is asserted.

Table 3–10 shows the configuration signals.

Table 3–10. Configuration Signals (Part 1 of 2)

Signal Name	Description
ber_clear	Reset for the BER counter. Only for the BER block option.
state_node_sync[log2(Nmax):1]	Specifies the node synchronization rotation to rr. The state_node_sync signal is latched when sink_sop is asserted.

Table 3–10. Configuration Signals (Part 2 of 2)

Signal Name	Description
sel_code[log2(Ncodes):1]	Selects the codeword—'0' selects the first codeword, '1' selects the second, and so on. The bus size increases according to the number of codes specified. sel_code is latched when sink_sop is asserted.
tb_length[]	Traceback length. The maximum width of tb_length is equal to the maximum value of parameter v. The tb_length input is latched when sink_sop is asserted. This signal is disabled if you select the continuous optimization and you must remove it from the testbench.
tb_type	Altera recommends that you set tb_type high always for future compatibility. In block decoding when tb_type is low, the decoder starts from state 0; when tb_type is high, the decoder uses the state specified in tr_init_state[(L-1):1]. For block decoding set tb_type high. tb_type is latched when sink_eop is asserted. If you select continuous optimization, this input is removed from the top level design and connected to zero in the inner core.
tr_init_state[(L-1):1]	Specifies the state to start the traceback from, when tb_type is asserted high. tr_init_state is latched when sink_eop is asserted. If you select continuous optimization, this input is removed from the top level design and connected to zero in the inner core. For more information, refer to "Trellis Termination" on page 3–8.
bm_init_state[(L-1):1] *(1)*	Specifies the state in which to initialize with the value from the bm_init_value[] bus. All other state metrics are set to zero. bm_init_state is latched when sink_sop is asserted.
bm_init_value[(L-1):1] *(1)*	Specifies the value of the metric that initializes the start state. All other metrics are set to 0. bm_init_value must be larger than $(L \times 2^{(softbits - 1)})$. bm_init_value is latched when sink_sop is asserted.

Note to Table 3–10:
(1) Hybrid architecture only.

Table 3–11 shows the status signals.

Table 3–11. Status Signals (Part 1 of 2)

Signal	Description
normalizations[8:1]	The normalizations bus indicates in real time the number of normalizations that have occurred since sink_sop was last activated. (Refer to "State Metrics" on page 3–2.)
numerr[] *(1)*	The numerr bus contains the number of errors detected during a block. It is updated each time an error is detected, so you can see the location of individual errors. It is reset when source_sop asserted; it is valid two-clock cycles after source_sop. IP Toolbench automatically sets the width of this bus. This signal is left open if you do not select a BER block.
bestmet[bmgwide:1]	The best metric. The bestmet signal shows the best state metric for every trellis step as it is being found by the best state finder. The state that contains this best metric is shown in bestadd. This signal is left open, if you select continuous optimization, or if you select none for optimization and turn off best state finder in IP Toolbench.

Table 3–11. Status Signals (Part 2 of 2)

Signal	Description
`bestadd[(L-1):1]`	The best address state. The address corresponding to the best metric as it is being found by the best state finder. The metric of this state if shown in `bestmet`. This signal is left open, if you select continuous optimization, or if you select none for optimization and turn off best state finder in IP Toolbench.

Note to Table 3–11:

(1) Used only when you select the BER estimator option.

Timing Diagrams

Figure 3–12 shows the hybrid Viterbi decoder input timing diagram. The `sink_rdy` signal is asserted for one clock cycle in every Z clock cycles. (For the values of Z, refer to "Latency Calculator" on page 3–14.) If the decoder becomes full because data is not being collected on the source side, it may deassert `sink_rdy` until it can accept new data. The decoder only accepts data, if `sink_rdy` is asserted.

Figure 3–12. Input Timing Diagram—Hybrid

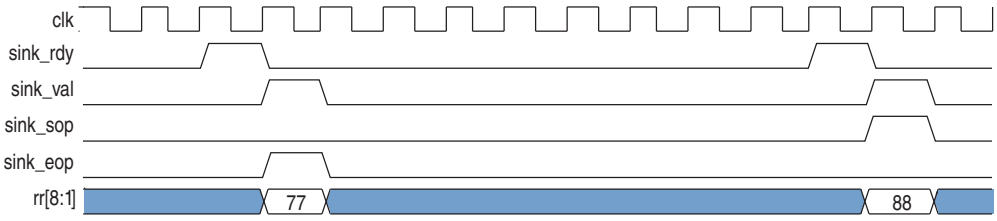


Figure 3–13 shows the parallel Viterbi decoder input timing diagram.

Figure 3–13. Input Timing Diagram—Parallel

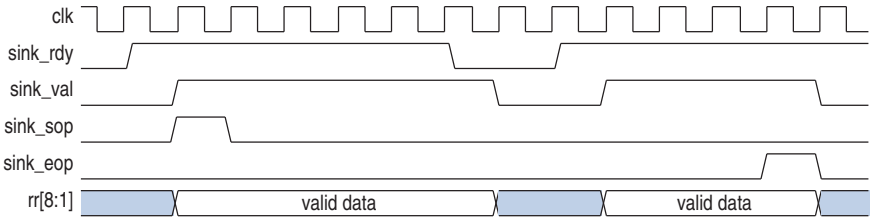


Figure 3–14 and 3–19 show output timing diagrams. Figure 3–14 shows the `source_val` signal asserted initially for 8 or 16 clock cycles. It is then asserted for the number of clock cycles corresponding to the amount of remaining data, if `source_rdy` remains asserted.

Figure 3–14 shows the typical ending of a block or packet in the Avalon-ST interface on the source (Viterbi) to the sink (user) side connection.

Figure 3–14. Output Timing Diagram—Example 1

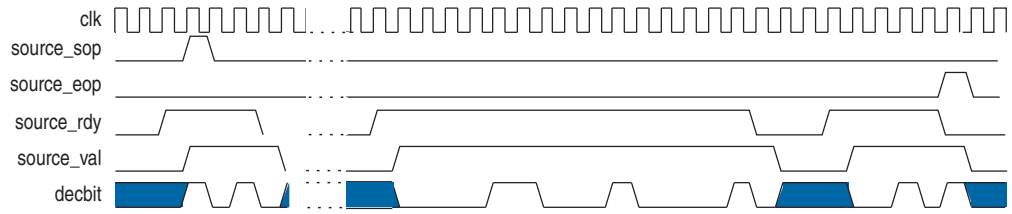


Figure 3–15 on page 3–19 shows a different ending.

Figure 3–15. Output Timing Diagram—Example 2

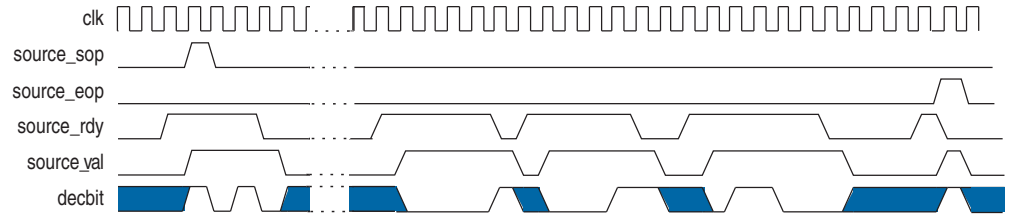
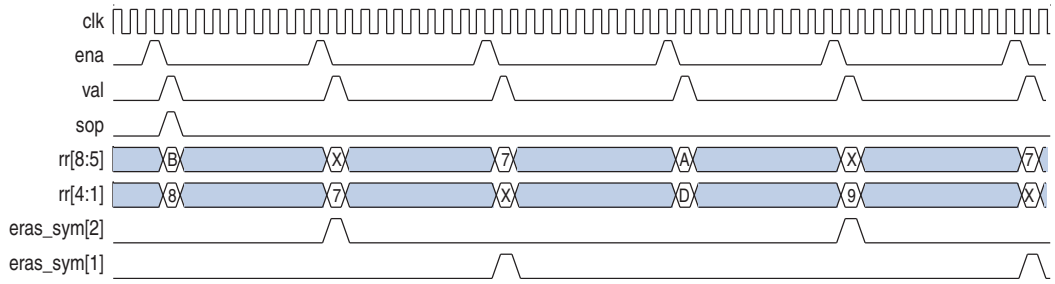


Figure 3–16 shows a depuncturing timing diagram and shows `eras_sym` for the pattern 110110 (puncturing rate 3/4). By changing the `eras_sym` pattern you can implement virtually any depuncturing pattern you require.

Figure 3–16. Depuncturing Timing Diagram



MegaCore Verification

The MegaCore verification includes an automated regression test suite, which is described in the following paragraphs.

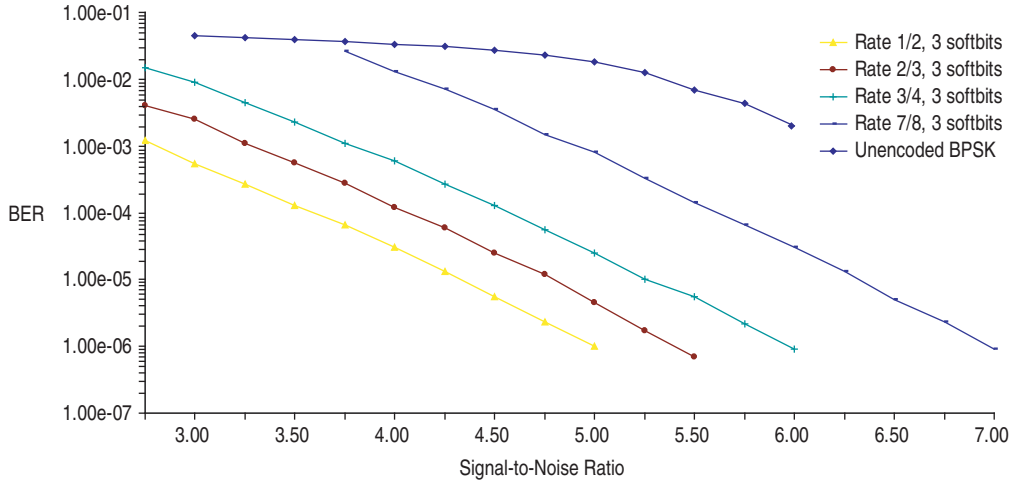
Scripts drive the simulation at RTL level. Data is randomly generated and encoded. The original transmitted bits are stored in a file **transbit.txt**. Optionally, Gaussian noise is added as a channel model and the data is formatted for use by the decoder's testbench. The file that feeds the testbench is **a_rcvsym.txt**. The testbench collects the decoder's decoded bits and stores them in **decoded.txt**. Those bits are compared with the original in **transbit.txt**.

A script defines sets of tests that cover a comprehensive set of parameters on RTL VHDL simulation.

The testbenches can generate many patterns for the Avalon-ST interface testing and all the possible scenarios are tested.

The first tests are carried out with noiseless data. Then tests using a subset of parameters, which use data with noise and performing millions of bits at different signal-to-noise ratios, are carried out to evaluate the BER performance. The BER performance matches the theoretical behavior of a Viterbi decoder (Figure 3–17 on page 3–20).

Figure 3–17. Graph of Actual BER vs. Signal-to-Noise Ratio for Various Values of Rate



Another subset of parameters are tested with noiseless data using post-synthesis Vital VHDL netlist.

The set of test patterns and parameters are comprehensive and should detect any malfunction in any of the features or parameter sets of the hybrid and parallel architectures.

This chapter provides additional information about the document and Altera.

Revision History

The following table shows the revision history for this user guide.

Date	Version	Changes Made
November 2012	12.1	Added support for Arria V GZ devices.
May 2011	11.0	■ Updated support level to final support for Arria® II GX, Arria II GZ, Cyclone® III LS, and Cyclone IV GX devices. ■ Updated support level to HardCopy Compilation for HardCopy III, HardCopy IV E, and HardCopy IV GX devices.
December 2010	10.1	■ Added preliminary support for Arria II GZ devices. ■ Updated support level to final support for Stratix IV GT devices.
July 2010	10.0	■ Added preliminary support for Stratix V devices.
November 2009	9.1	■ Maintenance update. ■ Added preliminary support for Cyclone III LS, Cyclone IV, and HardCopy IV GX devices.
March 2009	9.0	Added Arria II GX device support.
November 2008	8.1	Added notes to trellis termination and trellis initiation sections.
May 2008	8.0	Added device support for Stratix IV devices.
October 2007	7.2	Maintenance release.
May 2007	7.1	■ Added support for Arria GX devices. ■ Amended signal descriptions. ■ Added new `ber_clear` signal. ■ Added parallel architecture optimization options.
December 2006	7.0	Added support for Cyclone III devices.
December 2006	6.1	Updated format.

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Nontechnical support (general)	Email	nacomp@altera.com

Contact (1)	Contact Method	Address
(software licensing)	Email	authorization@altera.com

Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, *<file name>* and *<project name>***.pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections in a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix `n` denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
↵	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	The question mark directs you to a software help system with related information.
	The feet direct you to another document or website with related information.
	The multimedia icon directs you to a related multimedia presentation.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.